Salvar dados e reencontrá-los rapidamente: árvores Red-Black

Árvores coloridas

PROGRAMAÇÃO

O que os esquilos e o kernel Linux têm em comum? Ambos percorrem árvores para armazenar coisas e depois reencontrá-las. Mas nessa comparação o kernel Linux trabalha melhor – por exemplo ao lidar de maneira bem peculiar com árvores pintadas de duas cores.

por Tim Schürmann

O esquilo Sandy já não tem mais a juventude de antes. Não apenas os ossos lhe doem, como também a boa memória de antes o abandonou. No ano passado, quando procurava sua noz preferida guardada do ano anterior, ele foi obrigado a vasculhar entre todos seus pertences. Assim como na vida real, a guloseima estava escondida entre os últimos itens do estoque.

Muitos programadores certamente se identificarão com o sofrimento de Sandy. Assim como o velho esquilo, eles armazenam seus dados desordenadamente na memória. Um vetor ou inteiro longo servem como depósitos de ponteiros para listas interligadas, como por exemplo:

```
typedef struct _elemento{
   void *ponteiroparameusdados;
   struct_element *proximo;
} elemento;
```

O ponteiro `proximo` sempre se refere ao elemento seguinte na lista. Se agora o programador estiver procurando um elemento armazenado ali, ele precisa percorrer exaustivamente a complexa lista. Quanto mais elementos essa lista contiver, mais demorada será a busca – na pior das hipóteses, é preciso verificar cada um dos elementos. Se ainda inserirmos essa busca ineficiente em um laço, o tempo de execução de nosso código se alongará insuportavelmente.

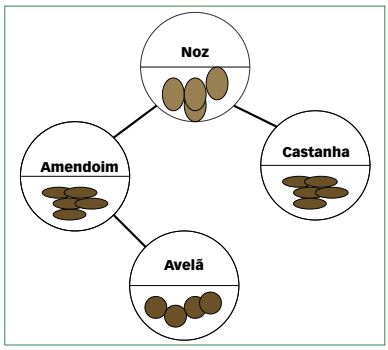

Figura 1: Um exemplo de árvore de busca que administra as nozes de Sandy. As chaves são os nomes dos tipos na parte superior de cada nó, enquanto o conteúdo relacionado está localizado no campo inferior.

Kernel

O kernel Linux enfrenta problemas semelhantes com freqüência. Por isso, o sistema operacional necessita, por exemplo, saber quais regiões da memória já estão ocupadas. Ele poderia arquivar essas informações em uma lista encadeada como a que descrevemos. Nesse caso, se o kernel quisesse determinar se uma região de salvamento ainda está livre, ele teria que, em algumas situações, percorrer longas seqüências, sempre até o fim. Outro exemplo é oferecido pelo escalonador de E/S: assim que um programa precisa acessar o disco rígido, ele envia uma solicitação ao kernel do sistema operacional.

Lá, o escalonador de E/S assume outras atividades. De maneira simplificada, ele primeiro memoriza todas as solicitações entrantes em uma fila de espera, para depois finalmente colocá-las em uma seqüência mais organizada. Isso deve acelerar o acesso ao disco rígido. Quanto mais solicitações um programa apresentar, mais dados terão que ser administrados pelo escalonador. Aqui, uma lista simples conseguiria prejudicar fortemente o desempenho de todo o sistema, e novamente devorar em parte a vantagem de aceleração.

Na árvore

Como o esquilo Sandy já está um pouco senil, mas ainda lúcido, ele vai implementar um sistema de armazenagem inteligente em seu depósito ainda este ano. Para isso, ele procurou uma bela árvore antiga com muitos galhos. Antes de guardar suas nozes ali, ele as mistura ao acaso. A informática já teve essa idéia há algum tempo. Em comparação à árvore real de Sandy, aqui felizmente existem alguns espaços sobrando, e com isso algumas vantagens.

Ela consiste de uma raiz e muitas bifurcações, chamadas "nós", que podem acomodar quaisquer dados. As hastes de conexão são chamadas "arestas". Os últimos nós da copa da árvore chamam-se, como os seus equivalente reais, "folhas". Ao contrário da vida do esquilo, no mundo da informática as árvores ficam de cabeça para baixo, as raízes normalmente são desenhadas em cima e o caminho para as folhas se estende para baixo.

Árvore binária

Para que fique tudo ainda mais fácil, nas próximas etapas cada nó deve exibir no máximo dois galhos, ou melhor, arestas. Em C, um nó como esse seria implementado da seguinte forma:

```
typedef struct _no{
   int data;
   struct _no *esquerda;
```

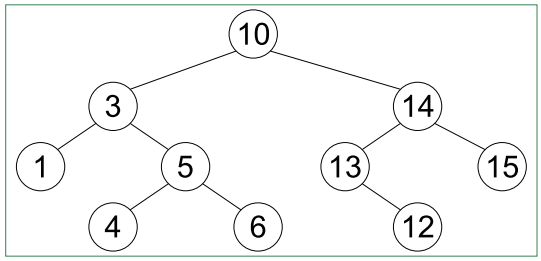

Figura 2: Uma busca pela chave 5 segue de acordo com este esquema: a busca se inicia na raiz da árvore binária. Como 5 é menor que 10, o próximo passo muda para o filho esquerdo. Aqui está o 3, que é menor que 5. Então a busca muda para o filho direito de 3, onde finalmente encontra-se o nó procurado.

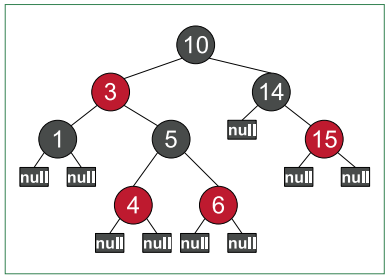

Figura 3: Exemplo de uma *Árvore Red-Black*. A raiz é sempre preta, enquanto os outros nós são vermelhos ou pretos.

```
   struct _no *direita;
   struct _no *pai;
} no
```

Para simplificar as coisas, suponhamos que os dados armazenados em um nó sejam números. É claro que em vez disso também é possível utilizarmos outros tipos de dados, desde que possam ser comparados de alguma forma. Em linguagem matemática: existe uma organização (parcial) para os nós.

Ambos os ponteiros apontam para o seu correspondente dos lados esquerdo e direito na seqüência de nós da árvore. Esses correspondentes se chamam filhos. Para depois tornar um pouco mais fácil a implementação dos algoritmos, a estrutura também salva os nós anteriores, que conseqüentemente são chamados de pais. Os nós da raiz são reconhecidos por possuírem a variável `pai` com valor `ZERO`.

Etiquetas

Sandy se encontra agora diante de uma pequena porção de nozes. Por motivos óbvios, ele colocou cada porção cuidadosamente em um pequeno saco e os nomeou de acordo com o tipo de noz que continham. Assim, nenhuma noz vai se misturar (**figura 1**). Agora Sandy consegue chegar ao conteúdo pela descrição dos pacotes.

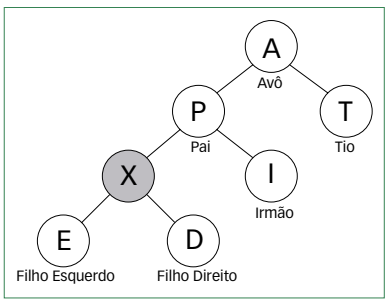

Figura 4: A relação de parentesco dos nós na representação gráfica. Todos os nós são parentes do nó "X".

De forma semelhante ao mundo dos bancos de dados, a descrição dos sacos se chama chave. De forma geral, o usuário gostaria de encontrar o mais rápido possível o elemento que corresponde à chave. Enquanto Sandy procura um determinado tipo de noz, o kernel procura o layout correspondente para uma região de armazenagem.

Uma estrutura de dados que solucione essa tarefa é conhecida pelos programadores pelo nome de dicionário ou *array* associativo. Não basta mais, portanto, salvar apenas as nozes, ou seja, os dados, em um nó. Em vez disso, as chaves mencionadas também precisam ser depositadas lá. Os exemplos seguintes tornam essa compreensão mais fácil, e utilizam, em vez de tipos de noz, novamente números crescentes.

Distribuidor

Em seguida, Sandy pensa sobre a melhor forma de distribuir os sacos de nozes pelos nós, para que a busca por um determinado tipo retorne o resultado sem o esquilo ter que escalar muito. Depois de refletir um pouco, lhe ocorre um sistema supostamente genial. Primeiro, ele fica parado próximo à raiz e olha para cima, o que corresponde a olhar para baixo na **figura 1**.

Agora ele pega um dos sacos e o coloca no primeiro nó. Finalmente, ele apanha todos os tipos de nozes que no alfabeto estão imediatamente antes daquele que acabou de ser estocado. Munido deles, ele se volta para a aresta esquerda, ou seja, a parte da árvore, e recomeça o jogo de outra forma. Por fim, ele repete o processo com o resto das nozes para a parte direita da árvore. Lá se encontram todas as nozes cujos nomes, no alfabeto, começam depois das nozes que se encontram na raiz.

Comparação de chaves

De maneira geral, para qualquer nó da árvore, todas as chaves da parte esquerda são menores, e todas as chaves do lado direito são maiores que a chave da raiz. A estrutura resultante é uma árvore de busca binária. Quem quiser reencontrar um elemento na árvore de busca deve iniciar pela raiz e olhar se a chave buscada é maior ou menor que a armazenada ali. Se for menor, a busca muda para a seqüência à esquerda da raiz; do contrário, para a direita. Isso dura até que seja encontrado um nó com a chave buscada, ou quando o nó não existir. Quando o nó correspondente é encontrado, significa que ele contém os dados procurados (veja a **figura 2**).

Baixa profundidade

Na necessidade de se administrar um grande volume de objetos e ter que localizá-los novamente da maneira mais rápida possível, esse tipo de árvore de busca binária é mais vantajosa em comparação com uma simples lista. Observando-se uma árvore com *n* nós, na qual cada nó tem exatamente dois filhos, um caminho vindo da raiz leva a uma folha através de, no máximo, $\log_2 n$ nós. A extensão desse caminho é chamada também de profundidade da árvore. Como se pode ver, mesmo no pior caso evita-se a necessidade de consultar todos os elementos armazenados.

Infelizmente, Sandy se deparou com um problema: ao colocar os montes de noz na seqüência crescente de nomes na árvore, ele obtêm uma lista muito longa. Então os nomes dos próximos tipos de noz sempre estão alfabeticamente após o que acabou de ser consultado. Conseqüentemente, ele segue sempre para a direita. Não resta mais nada ao pequeno esquilo, a não ser prestar atenção à arrumação das nozes e escolher uma distribuição adequada para os nós.

Felizmente, para os programadores isso é um pouco mais fácil. Eles podem simplesmente cortar qualquer aresta e colar os nós em qualquer lugar da árvore novamente. Essa técnica permite que, depois da inserção de um novo elemento, se evite essa formação de cadeia indesejada para, de alguma forma,

Exemplo 1: Tio e irmão

```
01 no* tio(no *n){
02    if (n->pai == n->pai->pai->esquerdo) return n->pai->pai->direito;
03    else return n->pai->pai->esquerdo;
04 }
05 no* irmao(no *n){
06    if (n == n->pai->esquerdo) return n->pai->direito;
07    else return n->pai->esquerdo;
08 }
```

trazer a profundidade novamente para o valor $\log_2 n$.

Muitas pessoas já tentaram solucionar esse problema de otimização. Em 1972, finalmente o professor de informática Rudolf Bayer foi bem-sucedido. Ele coloriu cada nó de uma árvore de busca binária de preto ou vermelho. Os algoritmos de otimização elaborados devem manter essas marcações coloridas das arestas com o mesmo nível.

O resultado atualmente é conhecido como árvore *Red-Black* (*Red-Black Tree*). Um nó dessa árvore em C tem a seguinte estrutura:

```
typedef struct _no{
   int chave;
   void *dados;
   enum { vermelho, preto } cor;
   struct _no *esquerda;
   struct _no *direita;
   struct _no *pai;
} no;
```

O escalonador de E/S no kernel Linux também recorre a diversas árvores red-black para as solicitações recebidas. Ele as utiliza como base para filas de espera especiais, nas quais, por exemplo, ele deposita cada solicitação de leitura que chega. As chaves podem ser, entre outras coisas, os setores requisitados do disco. Dessa maneira, o escalonador pode chegar a cada solicitação muito mais rapidamente, beneficiando assim o processo de otimização.

As árvores red-black também são empregadas no gerenciamento de memória. Elas ajudam na verificação rápida de regiões da memória e, com isso, na administração de espaços virtuais de endereçamento. Graças à árvore de buscas, o kernel consegue determinar rapidamente quais áreas estão ocupadas (quando a busca na árvore red-black é bem-sucedida) e quais ainda estão livres (em buscas sem sucesso).

Antes que os próximos parágrafos possam fornecer uma visão exata dos algoritmos de inserção e eliminação de elementos, é necessário um pequeno aviso: a implementação é trabalhosa. Por isso, uma implementação só é vantajosa quando grandes volumes de dados precisam ser administrados e localizados novamente. Quem tiver apenas cinco elementos para administrar muito provavelmente fará o serviço mais rápido com uma árvore de busca binária convencional, pois evitará todo o esforço de administração.

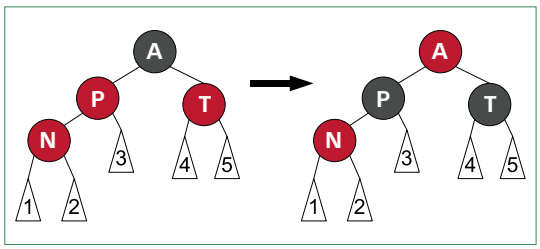

Figura 5: Inserção, caso 3: tio e pai são coloridos de preto, o avô de vermelho. A função de ordenação assegura que as condições das árvores *red-black* não serão infringidas.

Regulamentação

Uma árvore red-black é uma árvore de busca binária na qual cada nó possui a cor vermelha ou preta. Além disso, existem as seguintes condições:

- A raiz é sempre preta.
- As folhas da árvore são pretas e vazias; elas não armazenam, portanto, nozes ou qualquer outra informação. Elas apenas marcam o final da árvore. Em C, as folhas são implementadas através de um ponteiro nulo (a `esquerda` ou, conforme o caso, a `direita` do pai contém o valor `NULL`), e presumem que esses nós nulos são pretos (**figura 3**).

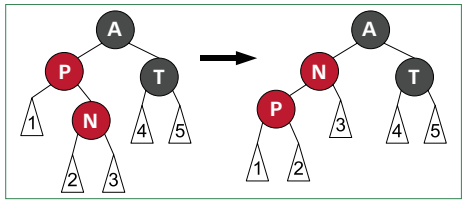

Figura 6: Inserção, caso 4: o novo nó *N* possui um pai vermelho e um tio preto. Caso ele esteja do lado esquerdo do pai, é necessária mais uma rotação direita no pai.

- Um nó vermelho tem apenas filhos pretos.
- Para cada caminho que parte da raiz até uma folha, o número de nós pretos percorridos é sempre igual.

A **figura 3** mostra o exemplo de uma árvore red-black. Ele preenche todos os quatro requisitos que logicamente também os algoritmos seguintes devem levar em consideração. A condição mais importante é a última. Ela faz com que a árvore não se degenere e não construa ramificações muito longas (o que também pode ser demonstrado de maneira formal). Antes de explicarmos sobre a inserção e exclusão de elementos, algumas correlações precisam ser esclarecidas. Depois de termos mencionado o relacionameto entre pai e filho, chegam agora o avô, o tio e o irmão. O avô é, como na vida real, o pai do pai (**figura 4**). No **exemplo 1** podemos ver como são definidos o tio e o irmão de um nó.

Uma busca funciona exatamente assim, como na árvore de busca binária. Aqui as cores dos nós não têm nenhuma importância e podem ser ignoradas. Esse procedimento é mostrado pelo **exemplo 2** em código C.

No entanto, inserir um novo nó é mais complicado. Primeiro procuramos um local apropriado para o novo elemento. Para isso, é indicado o algoritmo de busca normal. Como o elemento ainda não está contido na árvore, ele acaba indo parar em uma folha. Agora produzimos

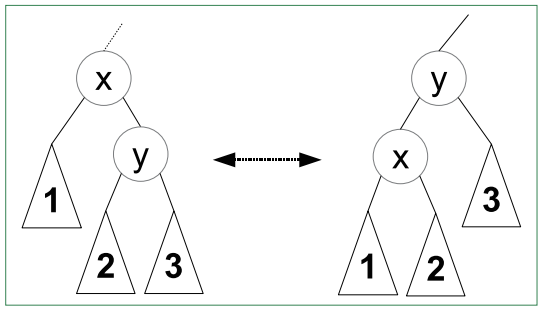

Figura 7: Uma rotação esquerda transforma a árvore da esquerda na da direita, uma rotação direita faz da árvore direita a representação daquela à esquerda.

Exemplo 2: Busca

```
01 no *busca(no *raiz, int chave)
02 {
03    // busca nos nós a chave "chave" na árvore de buscas com a raiz "raiz"
04
05    no *atual = raiz;
06    while (atual != NULL)
07    {
08       if (chave < atual->chave) atual = atual->esquerda;
09       else if (chave > atual->chave) atual = atual->direita;
10       else if (atual->chave == chave) return atual;
11    }
12    return NULL;
13 }
```

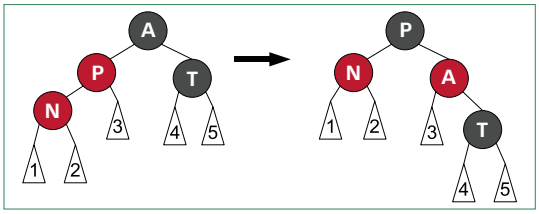

Figura 8: Inserção, caso 5: o novo nó *N* possui pai e tio vermelhos. Caso ele se encontre à esquerda do pai, é necessária ainda uma rotação à direita no avô.

um novo nó e o anexamos como novo filho na folha encontrada. Ao final, ele recebe uma coloração vermelha. Porém, agora uma das quatro condições poderia ser ferida. Vale a pena, portanto, verificar se é o caso de conduzir ações específicas de acordo com a situação.

Exemplo 3: Rotações

```
01 void rotacaoesquerda(no *n) {
02    no *x = n;
03    no *y = x->direita;
04    no *subarvore1 = x->esquerda;
05    no *subarvore2 = y->esquerda;
06    no *subarvore3 = y->direita;
07    no *paidex = x->pai;
08
09    if(paidex != NULL) {
10       if (n == paidex->esquerda) paidex->esquerda = y;
11       else paidex->direita = y;
12    }
13
14    y->pai = paidex;
15    y->direita = subarvore3;
16    y->esquerda = x;
17
18    x->pai = y;
19    x->esquerda = subarvore1;
20    x->direita = subarvore2;
21
22    if(subarvore1 != NULL) subarvore1->pai=x;
23    if(subarvore2 != NULL) subarvore2->pai=x;
24    if(subarvore3 != NULL) subarvore3->pai=y;
25 }
26 void rotacaodireita(no *n) {
27    no *y = n;
28    no *x = y->esquerda;
29    no *subarvore1 = x->esquerda;
30    no *subarvore2 = x->direita;
31    no *subarvore3 = y->direita;
32    no *paidey = y->pai;
33
34    if(paidey != NULL) {
35       if(n == paidey->esquerda) paidey->esquerda = x;
36       else paidey->direita = x;
37    }
38
39    x->pai = paidey;
40    x->esquerda = subarvore1;
41    x->direita = y;
42
43    y->pai = x;
44    y->esquerda = subarvore2;
45    y->direita = subarvore3;
46
47    if(subarvore1 != NULL) subarvore1->pai = x;
48    if(subarvore2 != NULL) subarvore2->pai = y;
49    if(subarvore3 != NULL) subarvore3->pai = y;
50 }
```

Caso a caso

A verificação e, caso necessário, o restabelecimento das condições, são assumidos pela função especial `ordenar()`, que recebe os nós transferidos e realiza todas as correções necessárias. Para isso, a função precisa escolher entre cinco casos possíveis:

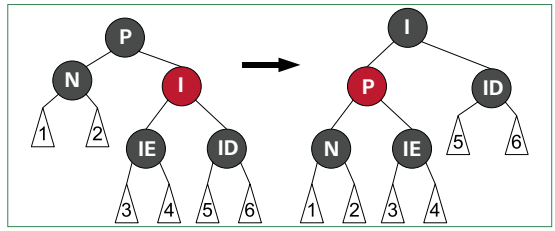

Figura 9: Exclusão, caso 3b: pai e irmão trocam de cor. Por fim, uma rotação direita restabelece a ordem.

- **Primeiro caso**: O novo nó é ao mesmo tempo também a raiz. Como não há qualquer outro nó na árvore, certamente a raiz pode ser pintada de preto sem problemas.
- **Segundo caso**: O novo nó tem um pai preto. Como o próprio novo nó é vermelho, todas as condições são satisfeitas, de modo que o processo já pode ser interrompido.
- **Terceiro caso**: O novo nó tem um pai vermelho e um tio vermelho, e com isso fere a condição 4. Nesse caso, simplesmente colorimos de preto o tio e o pai, e o avô de vermelho. Com isso, o avô poderia ferir uma das condições. Conseqüentemente os teste no bisavô devem prosseguir, nos quais a função `ordenar()` é agora aplicada a ele (**figura 5**).
- **Quarto caso**: (**figura 6**). O novo nó tem um pai vermelho e um tio preto. Se, além disso, ele for o filho direito de seu pai, segue-se uma chamada rotação à esquerda no pai. A **figura 7** mostra como isso funciona: a situação à esquerda é transportada para aquela à direita. Para isso, basta alterar o respectivo ponteiro do nó envolvido. Felizmente a rotação dispensa mais uma árvore de busca válida.

Caso o pai do novo nó seja o filho da direita do avô, segue-se, ao contrário da situação anterior, uma rotação simétrica à direita. Na **figura 7** isso corresponde à transferência da árvore da direita para a da esquerda. Independentemente da direção tomada pela rotação, finalmente, na visão do pai, se apresenta o quinto e ao mesmo tempo último caso, que ainda deve ser verificado. O **exemplo 3** mostra as funções para uma rotação para o nó `n`.

- **Quinto caso**: o novo nó tem um pai vermelho e um tio preto. Se ele, além disso, é o filho esquerdo de seu pai, segue-se uma rotação direita para o avô, conforme a **figura 8**. Se, no entanto, o novo nó é o filho direito, segue-se em vez disso uma rotação esquerda. Finalmente trocamos as cores entre pai e avô. O **exemplo 4** mostra o caso mais uma vez como código em C. O **exemplo 5** mostra o algoritmo de inserção completo em C.

Exclusão

Excluir nós da árvore red-black é ainda mais complicado que inseri-los. Primeiro o já conhecido algoritmo de busca procura pelo nó a ser eliminado. Quando encontrado, inicia-se novamente uma identificação do caso. O algoritmo completo – implementado em C – pode ser encontrado em [1].

- **Primeiro caso**: quando o nó a ser excluído possui dois filhos não-vazios, um sucessor deve ser definido. Ele pode ser o maior elemento na parte esquerda da árvore ou o menor elemento na parte direita da árvore (para entendidos de árvores: o *sucessor EmOrdem*). Agora o valor do sucessor deve ser transferido para o nó a ser eliminado. Com isso, o sucessor passa a ser irrelevante, e então pode ser excluído. Assim ele assume o papel do nó a ser excluído, para o qual os casos 2 e 3 ainda devem ser percorridos.
- **Segundo caso**: o nó a ser excluído possui duas folhas vazias como filhos (`esquerdo` e `direito` são ponteiros nulos). Nesse caso nomeamos uma das folhas vazias como sucessor. A

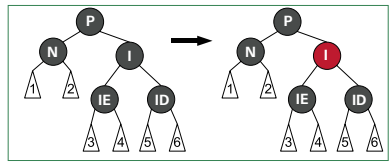

Figura 10: Exclusão, caso 3c. O irmão do novo nó é colorido de vermelho: como isso fere as condições da árvore red-black, inicia-se nele a função `ordenar()` mais uma vez.

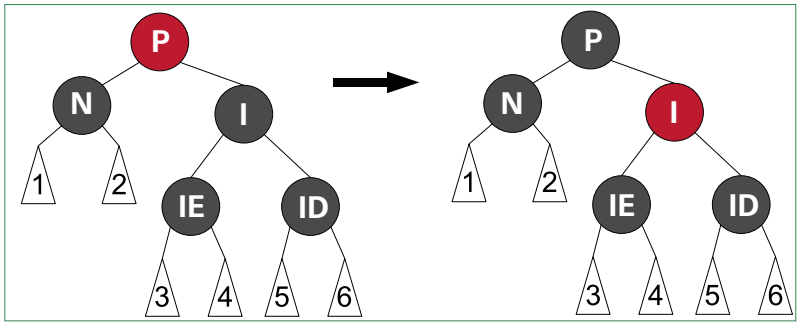

Figura 11: Exclusão, caso 3d (a variante fácil): basta o pai e o irmão trocarem a cor do nó a ser excluído.

outra folha está vazia de qualquer forma, e pode ser ignorada. Como isso se elimina o problema de exclusão do terceiro caso.

- **Terceiro caso**: O nó a ser excluído é exatamente o filho que não é um ponteiro nulo. Nesse caso, podemos primeiro tirar da árvore o nó a ser excluído e nomear o seu único filho como seu sucessor. Se o nó a ser excluído for preto e o sucessor vermelho, a condição 3, provavelmente violada, pode ser satisfeita, e com isso simplesmente colorimos o sucessor de preto.

No entanto, nos seguintes casos, depois de tomada essa ação, outras condições da árvore podem ser violadas (esses casos são apagados pela função auxiliar `ordenar_excluir()`), nos exemplos on-line [1]:

- O sucessor é a nova raiz. Como ele deve ser preto, não há mais nada a fazer.
- O novo irmão do sucessor é vermelho. Nesse caso trocamos as cores do pai e do irmão. Ao final, é realizada uma rotação à esquerda no pai, caso o sucessor seja o filho esquerdo de seu pai. Do contrário, segue-se uma rotação à direita (**figura 9**). Como resultado, a situação existente refere-se aos casos 3d, 3e ou 3f.
- O pai, o irmão e o filho do irmão são pretos. Nesse caso, o irmão recebe a cor vermelha, violando a condição 3 em relação ao pai. Para ele, a determinação do caso deve ser conduzida novamente com `ordenar_excluir()` (**figura 10**).
- O pai é vermelho, porém o irmão e o filho do irmão são pretos. Então basta trocar as cores entre o irmão e o pai (**figura 11**).
- O irmão e seu filho esquerdo são pretos, enquanto o filho direito do irmão é vermelho; e o próprio sucessor é o filho de seu pai. Primeiro trocamos as cores entre o irmão e seu filho direito.

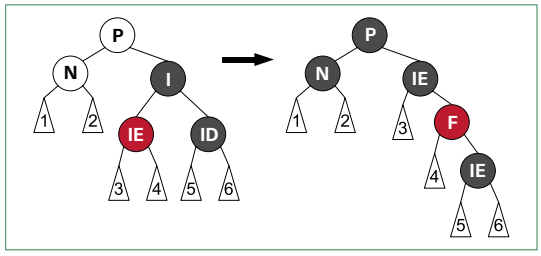

Figura 12: Exclusão, caso 3e. Um estado intermediário: no começo há a troca de cores entre irmão e filho direito. Depois de uma rotação à esquerda ou à direita, ocorre o caso 3f.

Exemplo 4: Ordenação

```
01 void ordenar(no *n)
02 {
03    no *dotio = tio(n);
04    no *avo = n->pai->pai;
05    no *pai = n->pai;
06
07    if (n->pai == NULL){
08       // 1o caso:
09       n->cor = preto; return;
10    }
11    else {
12       // 2o caso:
13       if (pai->cor == preto) return;
14       else {
15          // 3o caso:
16          if ((tio != NULL) && (tio->cor == vermelho)){
17             pai->cor = preto;
18             tio->cor = preto;
19             avo->cor = vermelho;
20             ordenar(avo);
21          }
22          else {
23             // 4o caso:
24             if( (n == pai->direita) && (pai == avo->esquerda) ) {
25                rotacaoesquerda(pai);
26                // transformar o antigo pai no no atual do 5o caso
27                n = n->esquerda;
28                tio = otio(n);
29                avo = n->pai->pai;
30                pai = n->pai;
31             } else if ((n == pai->links) && (pai ==avo->direita)){
32                rotacaodireita(pai);
33                // transformar o antigo pai no no atual do 5o caso
34                n = n->direita;
35                tio = otio(n);
36                avo = n->pai->pai;
37                pai = n->pai;
38             }
39
40             // 5o caso:
41             pai->cor = preto;
42             avo->cor = vermelho;
43             if((n == pai->esquerda) && (pai == avo->esquerda))
44                rotacaodireita(avo);
45             else rotacaoesquerda(avo);
46          }
47       }
48    }
49
50
51 }// Fim da ordenação
```

Exemplo 5: Inserção

```
01 void inserir(no *raiz, no* novono) {
      02   // Insere o nó novono na arvore cuja raiz é 'raiz'
      03   no *atual = raiz;
      04   no *antecessor = raiz;
      05   while (atual != NULL) {
      06     antecessor = atual;
      07     if (novono->chave < atual->chave) atual = atual->esquerda;
      08     else if (novono->chave > atual->chave) atual = atual->direita;
      09     else if (atual->chave == novono->chave) return;
      10   }
      11
      12   if (novono->chave < antecessor->chave) antecessor->esquerda = novono;
      13   else if (novono->chave > antecessor->chave) antecessor->direita = novono;
      14   novono->pai = antecessor;
      15   novono->cor = vermelho;
      16   ordenar(novono);
      17 }
```

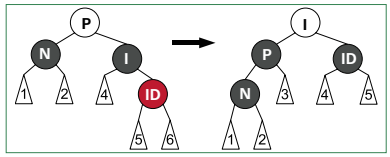

Figura 13: Exclusão, caso 3f. Dessa vez, irmão e pai trocam de cor. O centro da rotação a ser executada é o pai. O caso simétrico transcorre da mesma forma.

Finalmente rotacionamos à direita o irmão. A versão simétrica dessa situação transcorre de forma semelhante, porém à esquerda. Independentemente da direção, apresenta-se no final o caso 3f (**figura 12**).

- O filho direito do irmão é vermelho, o próprio irmão é preto e o sucessor é o filho esquerdo de seu pai. Nesse caso também trocamos novamente as cores, dessa vez entre irmão e pai. Por fim, rotacionamos à esquerda o pai. O caso simétrico segue de forma semelhante (**figura 3**).

Rápido, mas complicado

As árvores red-black aceleram consideravelmente a busca por elementos contidos nela, mas esse resultado não é facilmente atingível. Principalmente com grandes quantidades de dados, vale a pena o esforço de implementá-la. Quem tiver interesse em saber por que cada operação é conduzida em cada caso deve reproduzir os algoritmos em uma folha de papel para uma melhor compreensão.

Ao longo dos anos foram desenvolvidas diversas aplicações que deveriam reduzir o esforço de implementação e facilitar os algoritmos complexos. Por exemplo, o sistema de arquivos *ReiserFS* otimiza as árvores de buscas binárias caso os dados contidos nela estejam escritos no disco rígido.

Quem realmente pretender adotar as árvores red-black pode lançar mão de algumas implementações prontas, como por exemplo a *libredblack* [2]. Apesar disso, aconselha-se naturalmente ter um conhecimento básico do algoritmo a ser aplicado. ■

Mais Informações

[1] Exemplos on-line: http://www.linuxmagazine.com.br/issue/27/redblack.c

[2] Libredblack: http://libredblack.sourceforge.net